

Session 12

Performing Tests

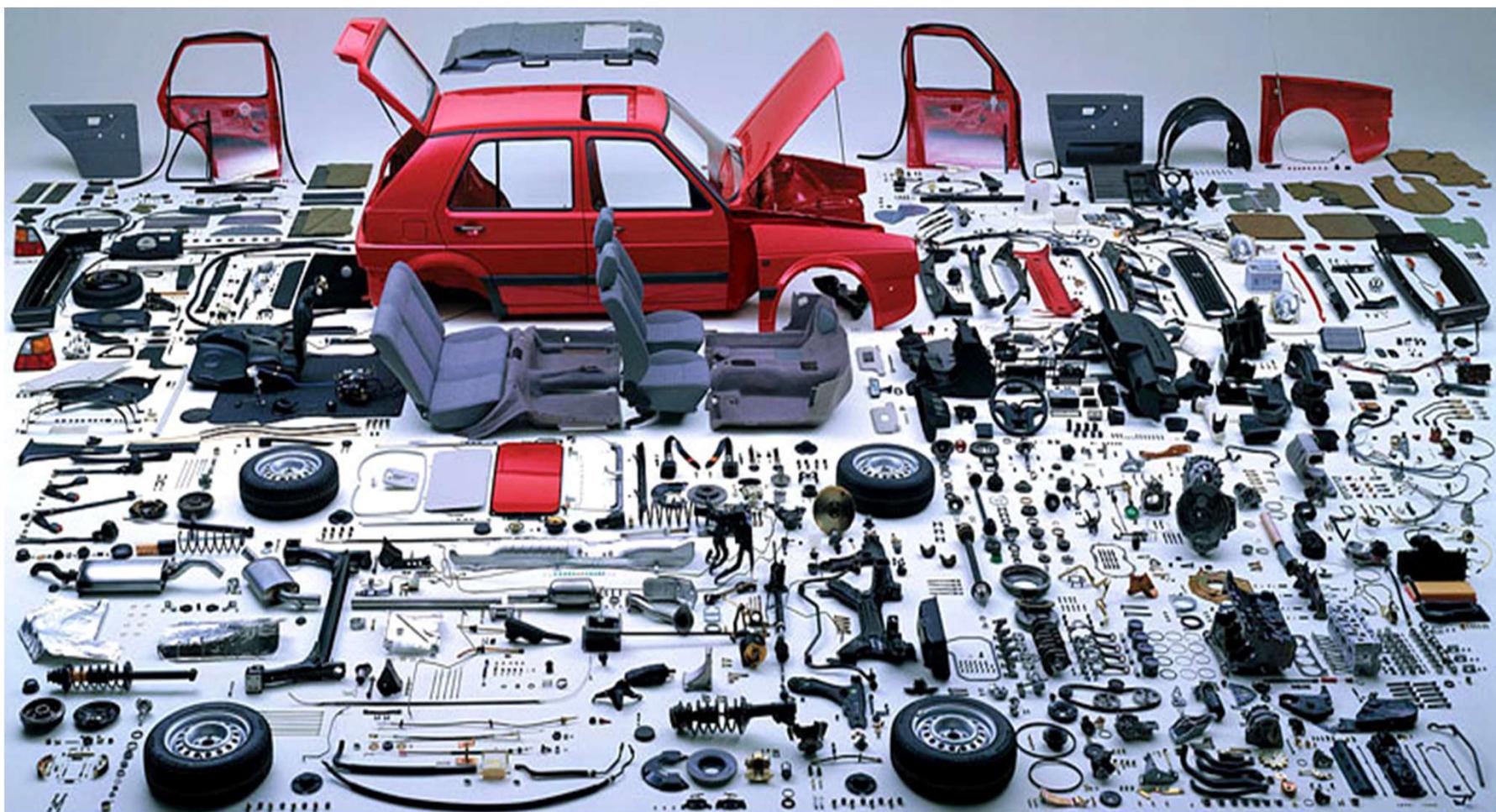
xfzhang@suda.edu.cn

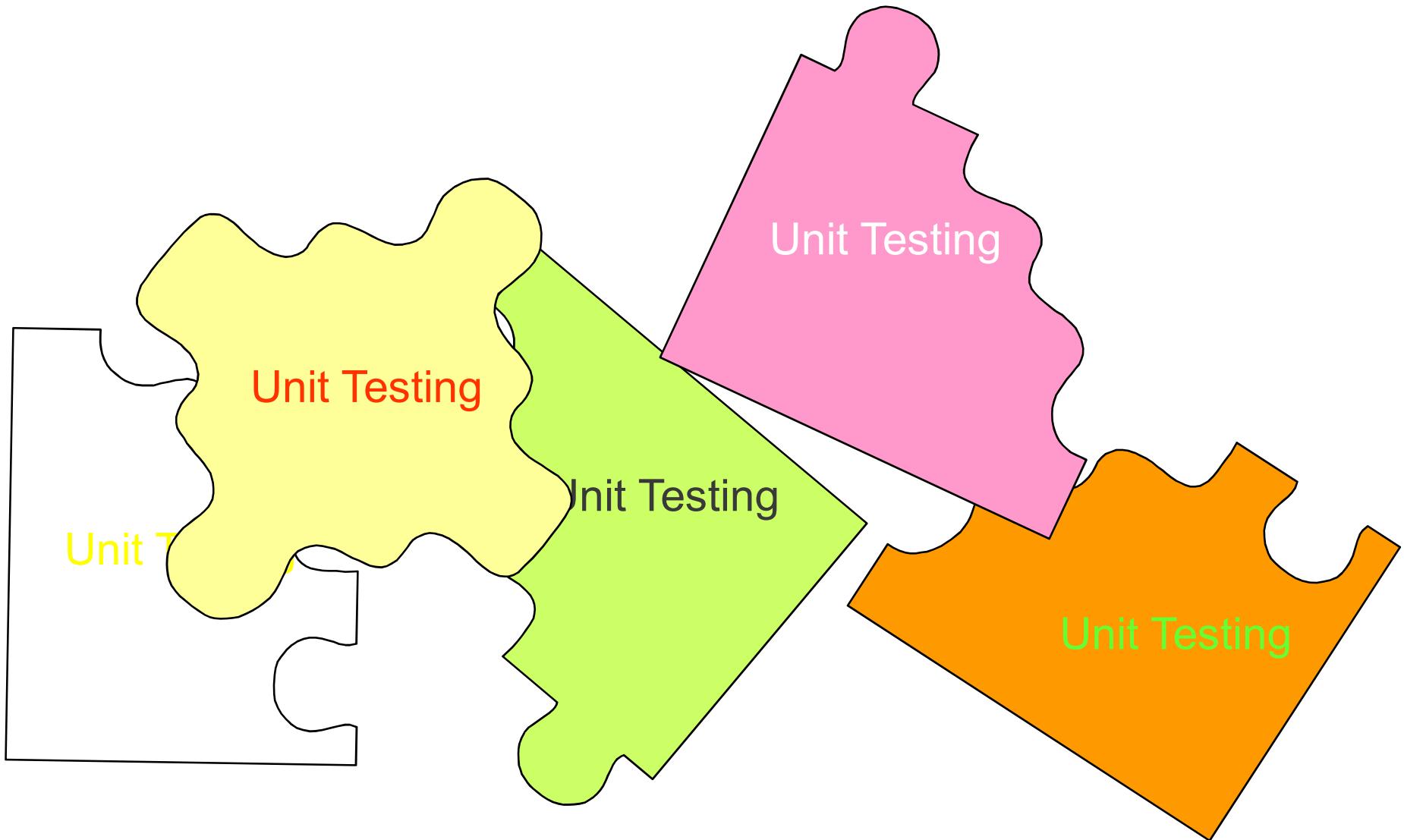
In this session, you will learn:

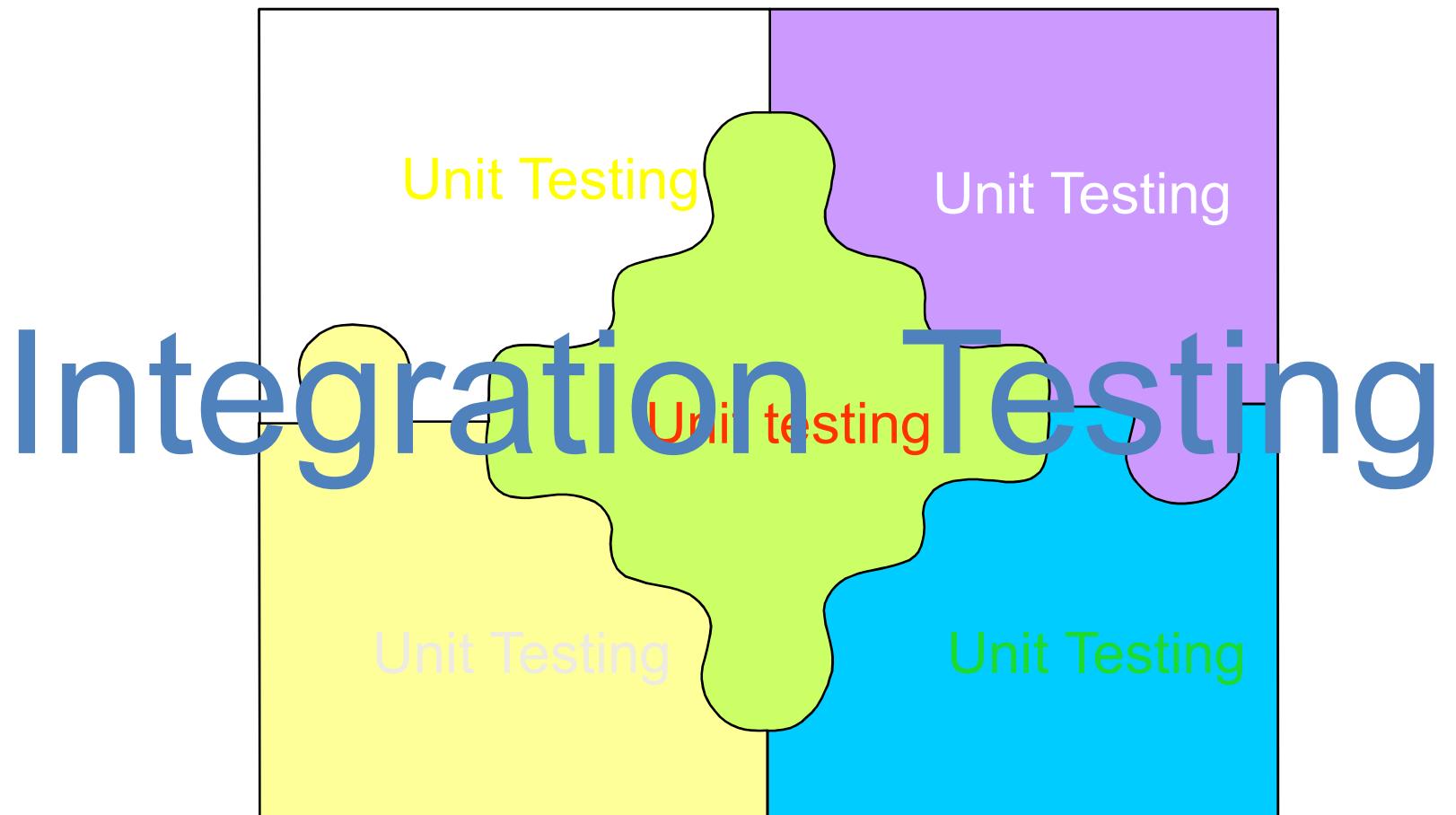
- Unit testing
- Integration testing
- System testing
- Acceptance testing
- Regression testing

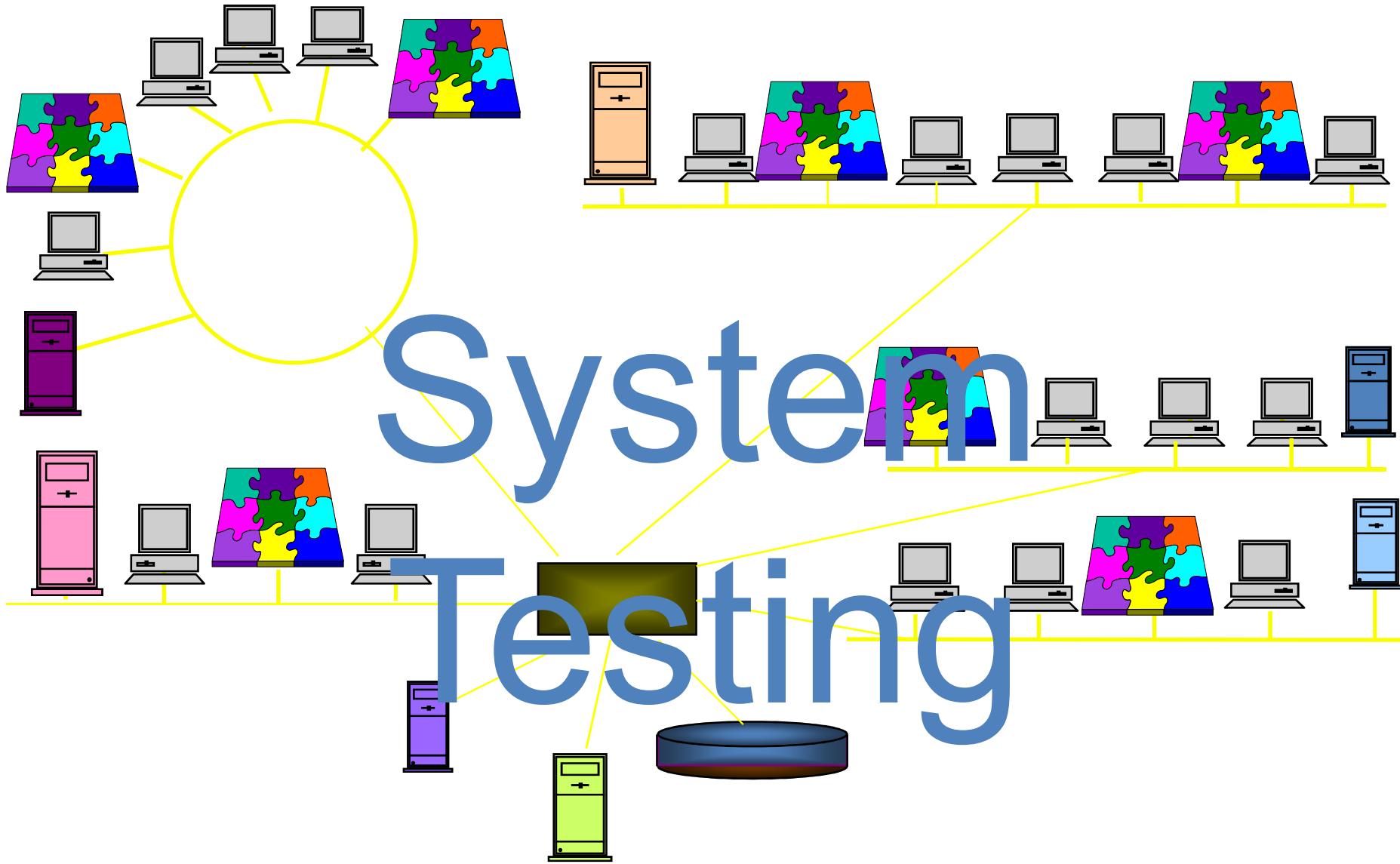
- Most **development teams** take the incremental view of testing.
 - Unit testing
 - Verification of isolated software units
 - Integration testing
 - Verification of the interaction among software units
 - System testing
 - Verification of the behavior of a whole system
- **Customers:** acceptance testing

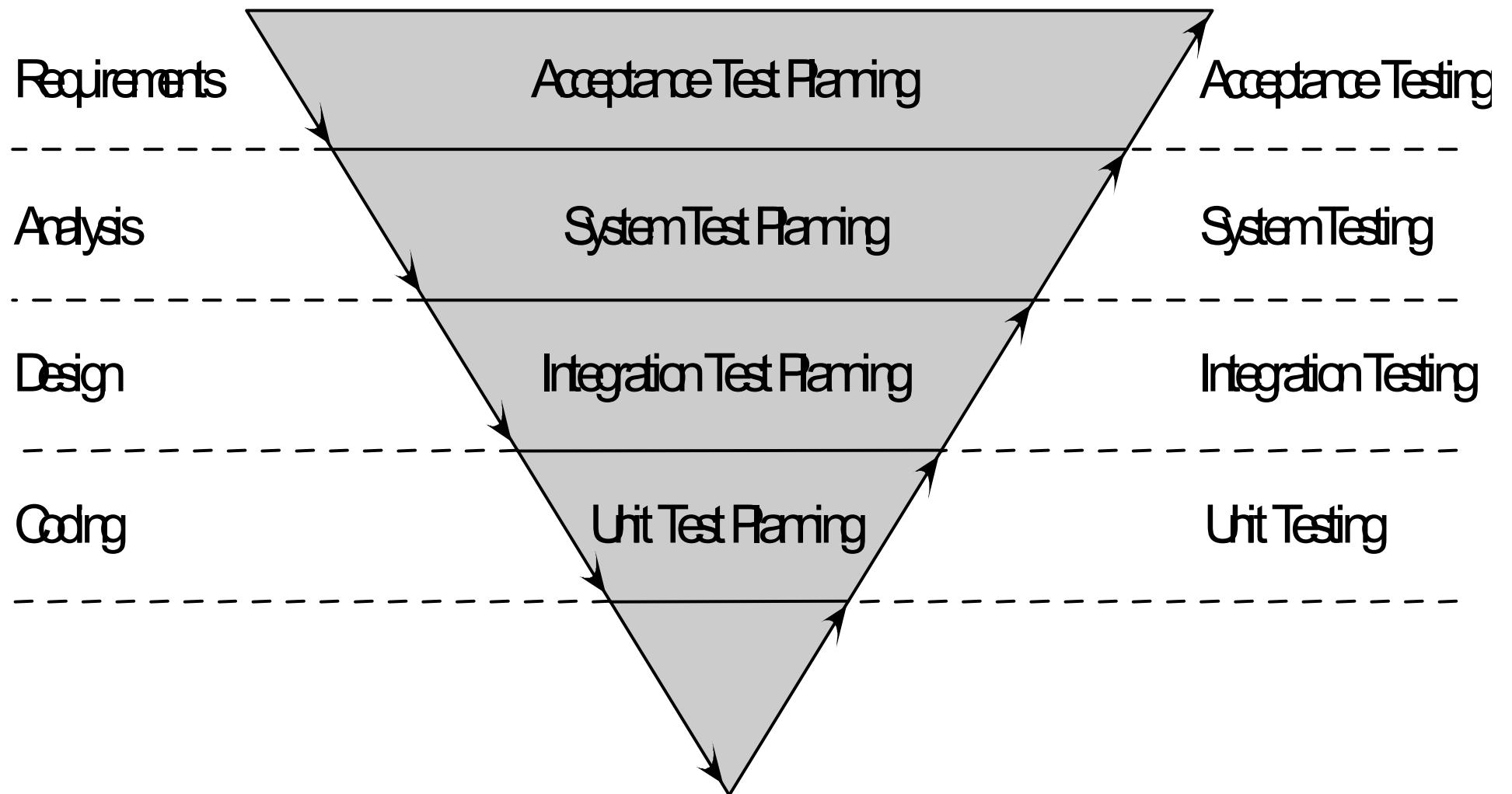
U-I-S-A











- Unit testing
- Integration testing
- System testing
- Acceptance testing
- Regression testing

5W+1H

Why we need Unit Testing

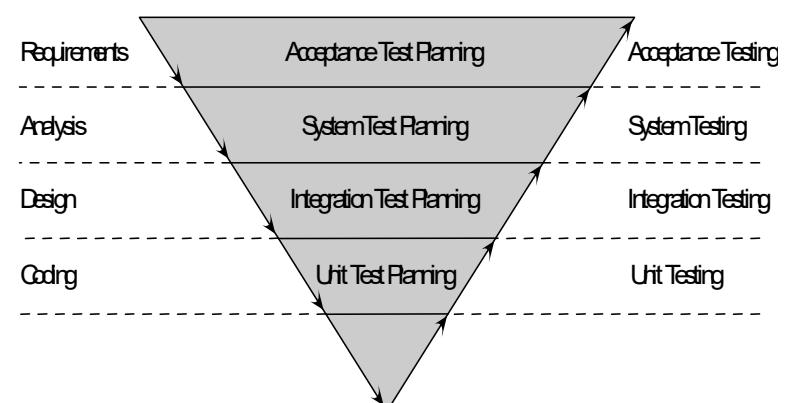
- In the programming process, dozens of errors are made for every 1000 lines of code 
- About **2-6 bugs** remain in every 1000 lines of code after compilation
- The cost of testing and debugging accounts for 30%-60% of overall development cost
- The earlier, the better

What is Unit Testing

- **Unit testing** is a software development process in which the **smallest testable parts** of an application, called units, are individually and independently checked for proper operation.
- A Unit may be
 - a program, a function, a procedure, a method, etc.

What is Unit Testing

- Purpose of unit testing:
 - Discover the errors introduced during **coding process**.
 - Validate whether code is consistent with the design.
 - Trace the implementation of requirements and design.
 - Discover the errors among design and requirement.



How to do Unit Testing

- Unit Testing
 - Static testing
 - It is primarily syntax checking of the code and/or manually reviewing the code or document to find errors.
 - This type of testing can be used **by the developer** who wrote the code, in isolation.
 - Peer reviews(同行评审), walkthroughs(走查) and inspections(审查) are also used.
 - Dynamic testing
 - Design and execute test cases
 - Tools: Junit, C++ Test, unittest(PyUnit), Pytest...

Peer reviews(同行评审)

- 一次检查少于200~400行代码
- 努力达到一个合适的检查速度：
300~500LOC/hour
- 有足够的时间、以适当的速度、
仔细地检查，但不宜超过60~90
分钟
- 在复审前，代码作者应该对代码
进行注释
- 使用检查表（checklist）能改进
双方（作者和复审者）的结果
- 验证缺陷是否真正被修复



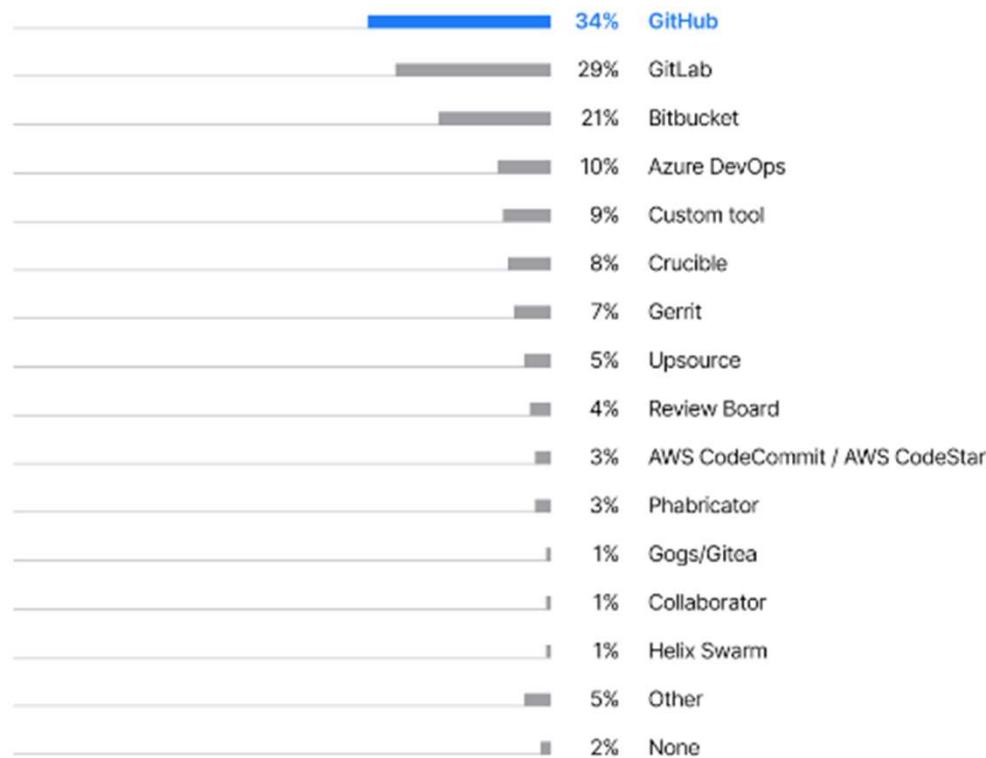
Comment: 接口、数据成员、
实现和模块依赖

Commit message generation

Peer review tool

根据《开发者生态系统现状》报告选出的最佳代码审查工具

Which tools do you use in your company / organization for Code Review?



Peer review tool

The screenshot shows a peer review interface. On the left, a red-bordered box contains a comment from BD: "Is this really sufficient? From what I understand, localCheckouts is null only when the changelist is not pending." Below it is another comment from JC: "I thought so too, but it turns out localCheckouts can also be null if there are no files attached to it at all, so we can get here with that." A third comment from BD follows: "But that's bad. We probably depend on localCheckouts being non-null (but empty) with pending changelists without files!" At the bottom of this box are buttons for "Accept", "Add Comment", "Create Defect", and "Mark as Read". A "Comment:" field contains a response from another user: "I don't think so; I checked all the references to getLocalCheckouts () and everyone seems to check for null." A "Submit Comment" button is at the bottom of this field. To the right, a Java code editor displays a snippet of code with several lines highlighted in yellow. The code includes an if-statement checking if localCheckouts is null or empty, and error handling code for pending changelists.

```
'  
}  
if ( ! foundInLocalList )  
{  
    // This might be because we don't have the rig  
    // also be because the changelist has no files  
    // in particular. Cases 7191, 11015.  
IScmLocalCheckout[] localCheckouts = changelis  
if ( localCheckouts == null || localCheckouts.  
{  
    System.err.println();  
    System.err.println( "ERROR: Changelist " +  
        return false;  
}  
  
// Currently the only other explanation is a  
System.err.println();  
System.err.println( "ERROR: Pending changelist "
```

CodeCollaborator

Walkthroughs(走查)

定义：采用讲解、讨论和模拟运行的方式进行的查找错误的活动。

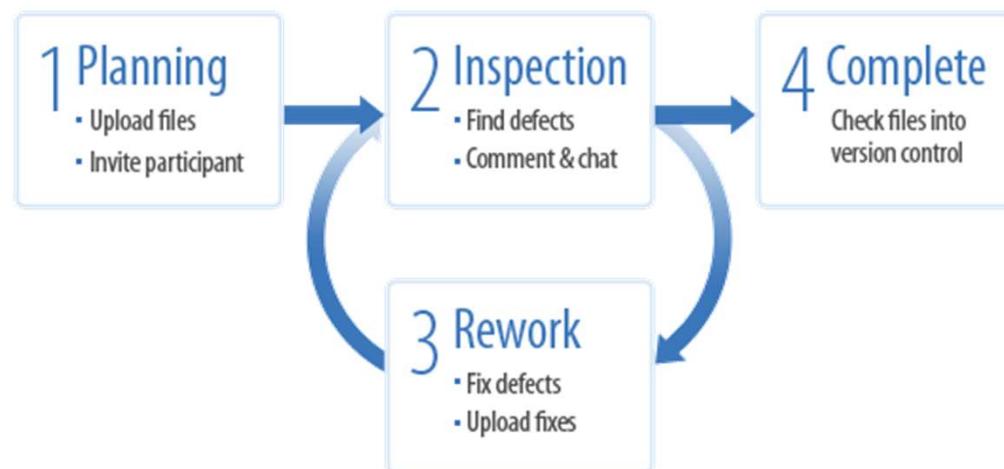
注意：

- 引导小组成员在走查前通读设计和编码。
- 限时，避免跑题
- 发现问题适当记录，避免现场修改
- 检查要点是代码是否符合标准和规范，是否有逻辑错误

Inspections(审查)

- 以会议形式，制定目标、流程和规则
- 按**缺陷检查表**（不断完善）逐项检查
- 发现问题适当记录，避免现场修改
- 发现重大缺陷，改正后会议需要重开

The Four Phases



Walkthroughs vs. Inspections

	走 查	审 查
准备	通读设计和编码	事先准备Spec、程序设计文档、源代码清单、代码缺陷检查表等
形式	非正式会议	正式会议
参加人员	开发人员为主	项目组成员包括测试人员
主要技术方法	模拟运行	缺陷检查表
生成文档	会议记录	静态分析错误报告
目标	代码标准规范 无逻辑错误	代码标准规范 无逻辑错误

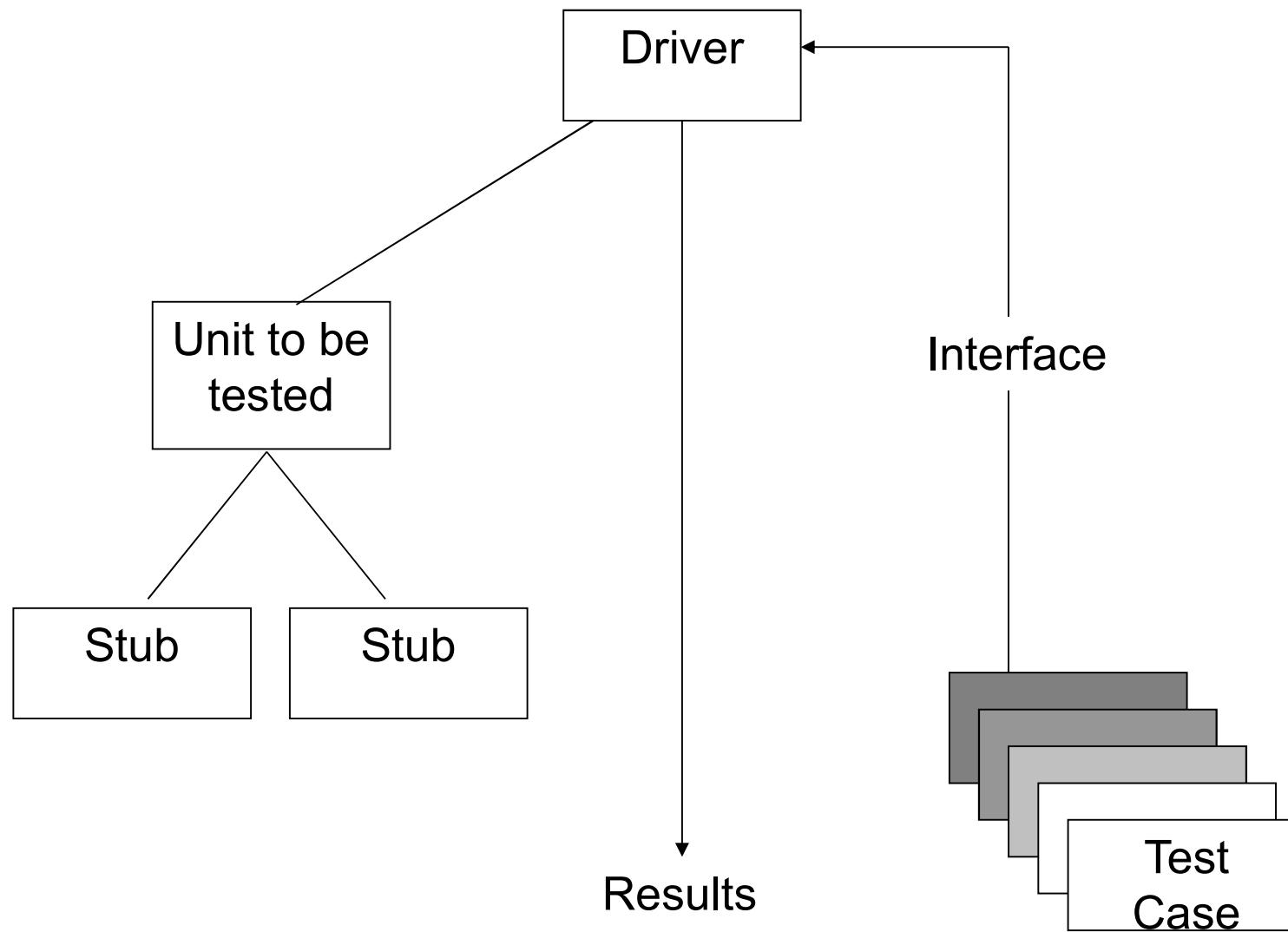
示例：单元测试检查表

关键测试项是否已纠正

1. 有无任何输入参数没有使用？有无任何输出参数没有产生？
2. 有无任何数据类型不正确或不一致？
3. 有无任何算法与PDL或功能需求中的描述不一致？
4. 有无任何局部变量使用前没有初始化？
5. 有无任何外部接口编码错误？即调用语句、文件存取、数据库错误。
6. 有无任何逻辑路径错误？
7. 该单元是否有多入口或多个正常的出口？

额外测试项

8. 该单元中有任何地方与PDL与PROLOG中的描述不一致？
9. 代码中有无任何偏离本项目标准的地方？
10. 代码中有无任何对于用户来说不清楚的错误提示信息？
11. 如果该单元是设计为可重用的，代码中是有可能妨碍重用的地方？



Unit Test Environment

Unit Testing Strategies

- **Driver:** is used to simulate **superior** module of tested module. It receives testing data, transmits related data to tested module, starts tested module and prints corresponding results.
- **Stub:** is used to simulate the **calling** modules in tested module. Generally, they only process few data.

```
1 //Program 1
2 #include <iostream.h>
3 void get_input(x& cost, int& y);
4 int main( )
5 {
6     double a;
7     int b;
8     char ans;
9     do
10    {
11        get_input(a, b);
12        cout.setf(ios::fixed);
13        cout.setf(ios::showpoint);
14        cout.precision(2);
15        cout << "a is " << a << endl;
16        cout << "b is " << b << endl;
17
18        cout << "Test again?"
19            << " (Type y for yes or n for no): ";
20        cin >> ans;
21        cout << endl;
22    } while (ans == 'y' || ans == 'Y');
23    return 0;
24 }
25 //Program 2
26 void function_under_test(int& x, int& y) {
27     ...
28     p = price(x);
29     ...
30 }
31 double price(int x) {return 10.00;}
```

Driver

Function under test

Stub

前置条件和后置条件

前置条件示例：

Contract.Requires(x != null);

后置条件示例：

Contract.EnsuresOnThrow<T>(this.F > 0);

Contract.Result<T>()

<http://msdn.microsoft.com/zh-cn/library/dd264808.aspx>

```
public string ReadAllText(string path)
{
    // Pre-conditions...
    if (path == null)
        throw new ArgumentNullException(...);
    if (path.Length == 0)
        throw new ArgumentException(...);
    if (IsOnlyWhitespace(path) ||
        ContainsInvalidCharacters(path))
        throw new ArgumentException(...);
    if (path.Length >
        GetMaximumFilePathLength())
        throw new PathTooLongException(...);

    // Open file and read and return contents
    // as string...
    object contents = GetFileContents(path);

    // Post-conditions
    if (!contents is string)
        throw
            new InvalidFileTypeException(...);
    if (string.IsNullOrEmpty(
        (string) contents))
        throw new EmptyFileException(...);
}
```

单元测试常用工具简介

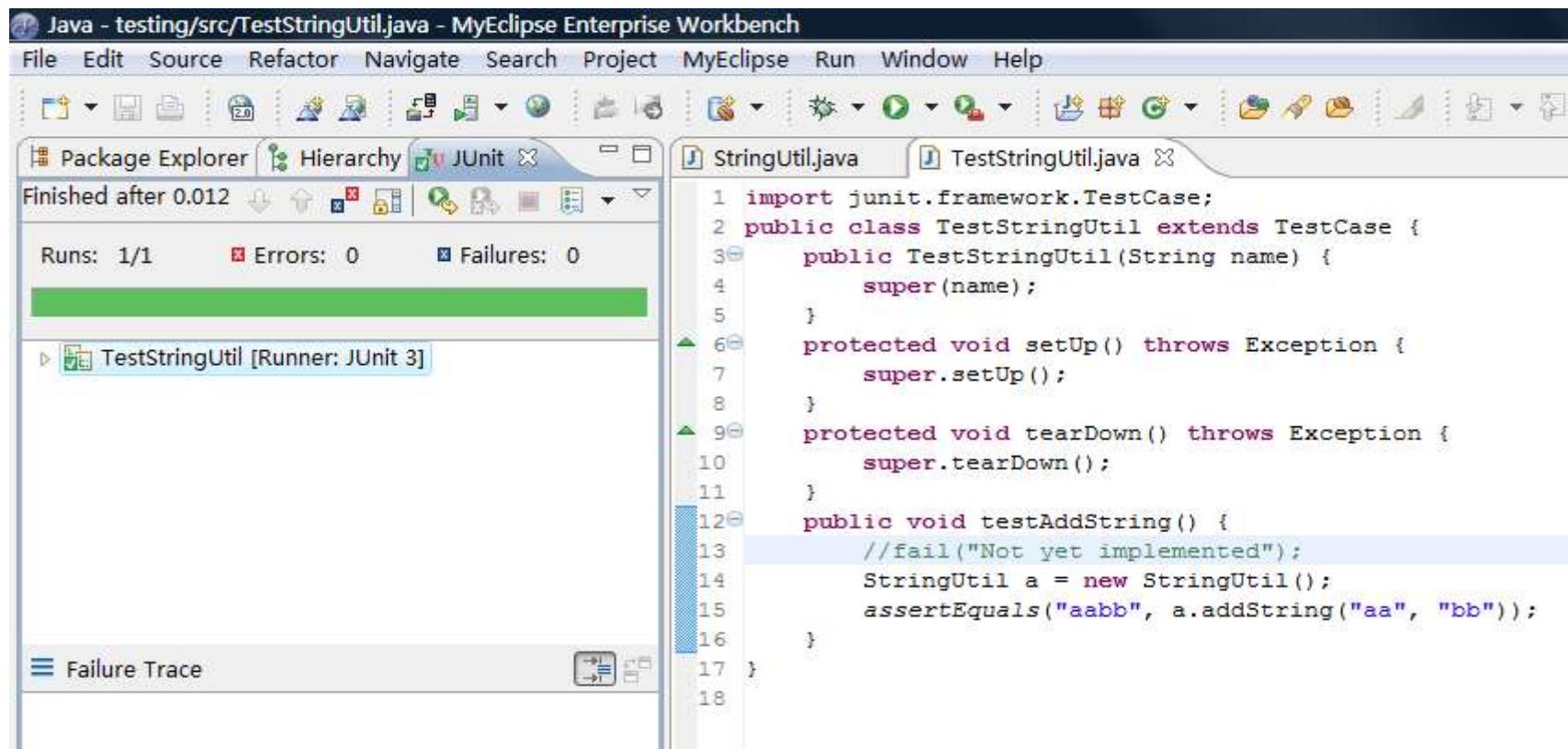
- 代码规则/风格检查工具
- 内存资源泄漏检查工具
- 代码覆盖率检查工具
- 代码性能检查工具

单元测试工具列表

类别	工具	朱少民《软件测试》
C 语言	C++ Test、CppUnit、QA C/C++、CodeWizard、Insure++6.0	朱少民《软件测试》
Java 语言	Jtest、Junit、Jmock、EasyMock、MockRunner	朱少民《软件测试》
JUnit 扩展框架	TestNG 、 JWebUnit 和 HttpUnit	朱少民《软件测试》
GUI (功能) 测试	JFCUnit、Marathor	朱少民《软件测试》
通用的	Rexelint、Splint、McCabe QA、CodeCheck、GateKeeper	朱少民《软件测试》
.NET	.TEST、Nunit	朱少民《软件测试》
Data Object, DAO	DDTUnit, DBUnit	朱少民《软件测试》
EJB	MockEJB 或者 MockRunner	朱少民《软件测试》
Servlet, Struts	Cactus, StrutsUnitTest	朱少民《软件测试》
XML	XMLUnit	朱少民《软件测试》
内嵌式系统等	Logiscope、JTestCase	朱少民《软件测试》

2021年软件测试工具总结——单元测试工具 (zhihu.com)

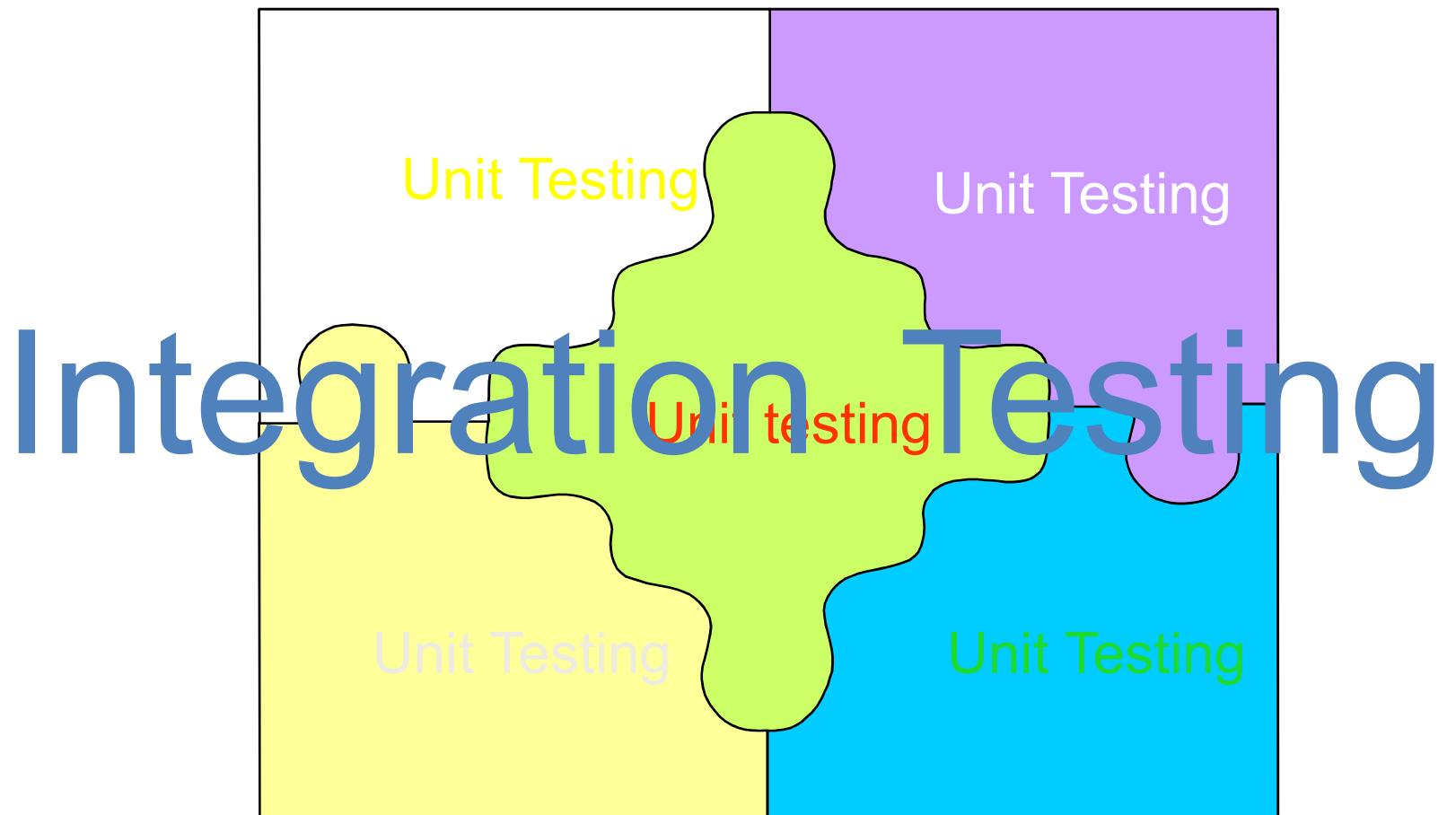
在Eclipse中JUnit应用举例



<http://www.junit.org/apidocs/overview-summary.html>

- 全国大学生软件测试大赛
- 第十五届蓝桥杯全国软件和信息技术专业人才大赛—软件赛—软件测试
- 软件测试能力认证联盟：腾讯、阿里巴巴、华为、软通动力、北京大学、南京大学和同济大学

- Unit testing
- Integration testing
- System testing
- Acceptance testing
- Regression testing



Integration Testing Introduction

- **Integration testing** (集成测试)
 - It is the phase of software testing in which individual software modules are combined and tested as a group.
 - Integration testing focuses on testing multiple components that work together.

Integration Testing Strategy

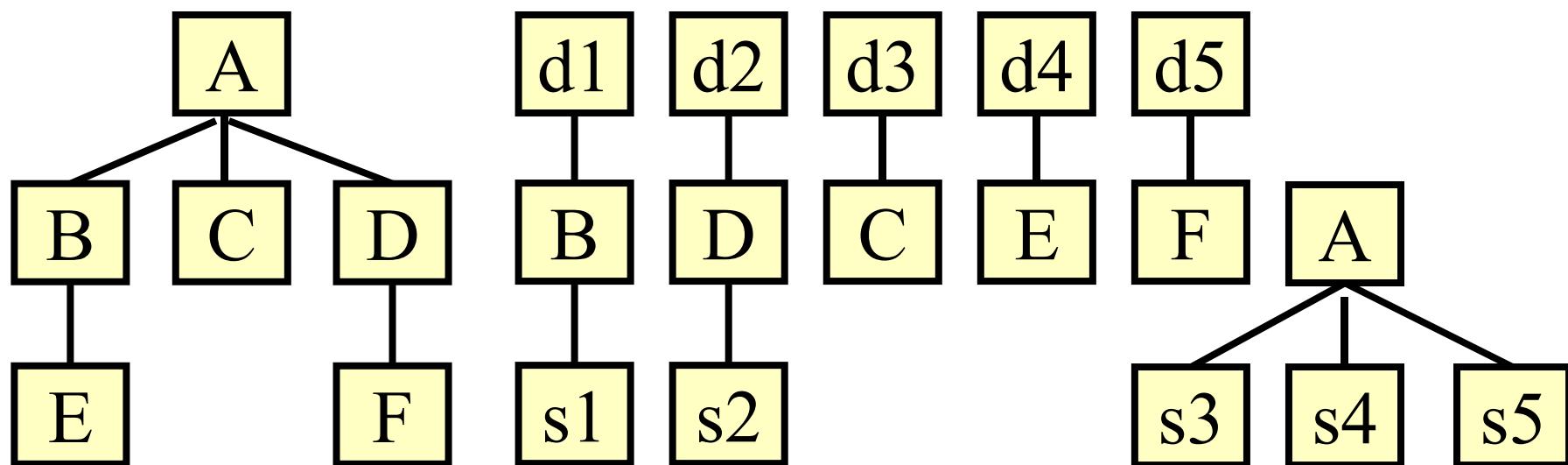
- Describe module integration (assembly) approach.
- How to combine the modules of the system? All at the same time assembly or gradually assembled?
 - The non-incremental integration strategy -- in one go
 - Incremental integration strategy -- gradually realize

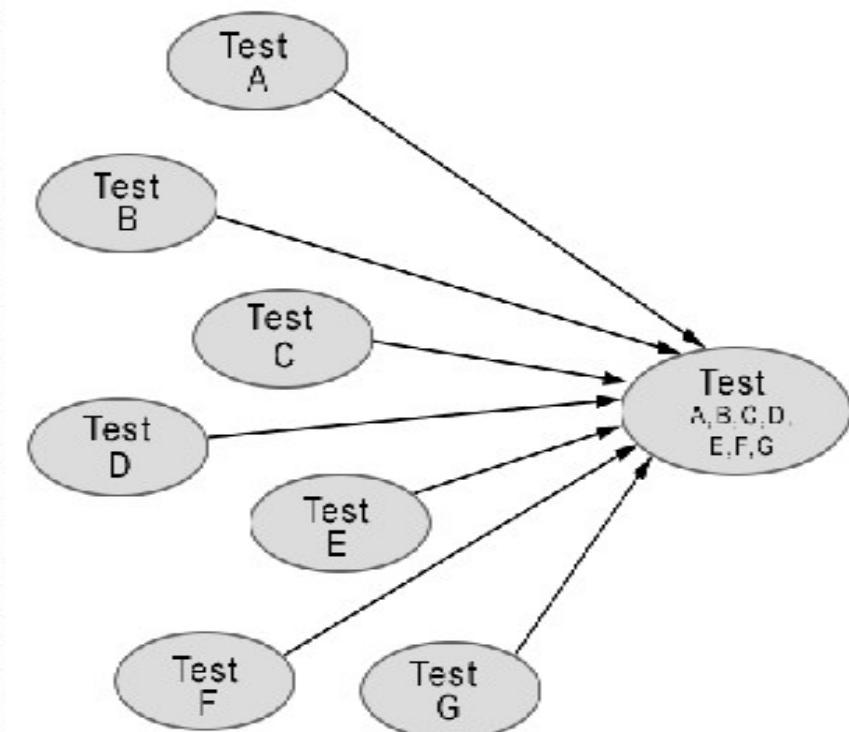
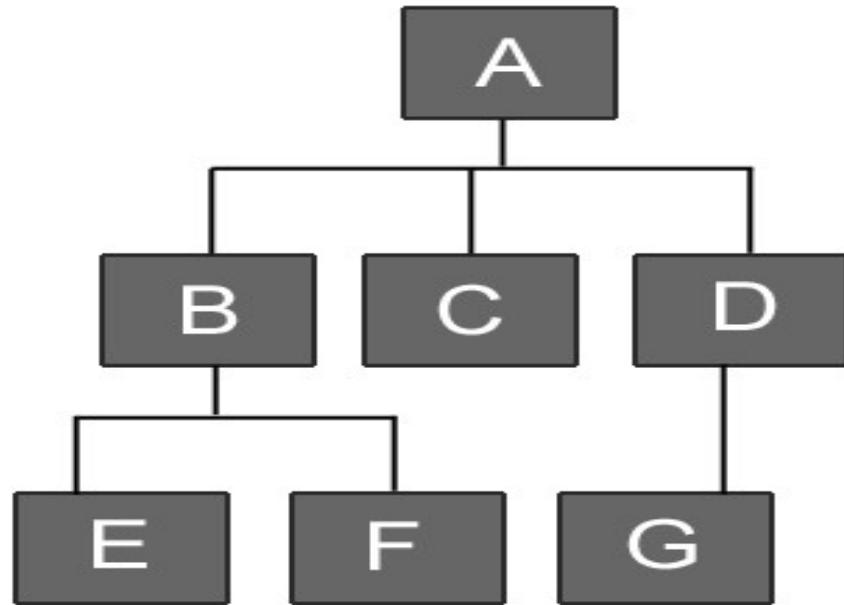
Integration Testing Strategy

- 基于分解的集成
 - Big bang (大爆炸) integration
 - Top-down (自顶向下) integration
 - Bottom-up (自底向上) integration
 - Sandwich (三明治) integration

Integration Testing Strategy

- Big Bang integration
 - It is a **non-incremental** integration method, it integrates all system components together **at one time**, not considering the dependence of components and the possible risk.
- Strategy:





先对每一个子模块进行测试（单元测试阶段）

然后将所有模块一次性的全部集成起来进行集成测试。

Integration Testing Strategy

- Advantage
 - Integration testing can be completed **rapidly** and only few stubs and drivers are needed.
 - Several testers can work in the **parallel** way, and human and material resources utilization is higher.
- Disadvantage
 - **localization and debug** more difficultly when error found.
 - Many **interface errors** can not be found until system testing.

Integration Testing Strategy

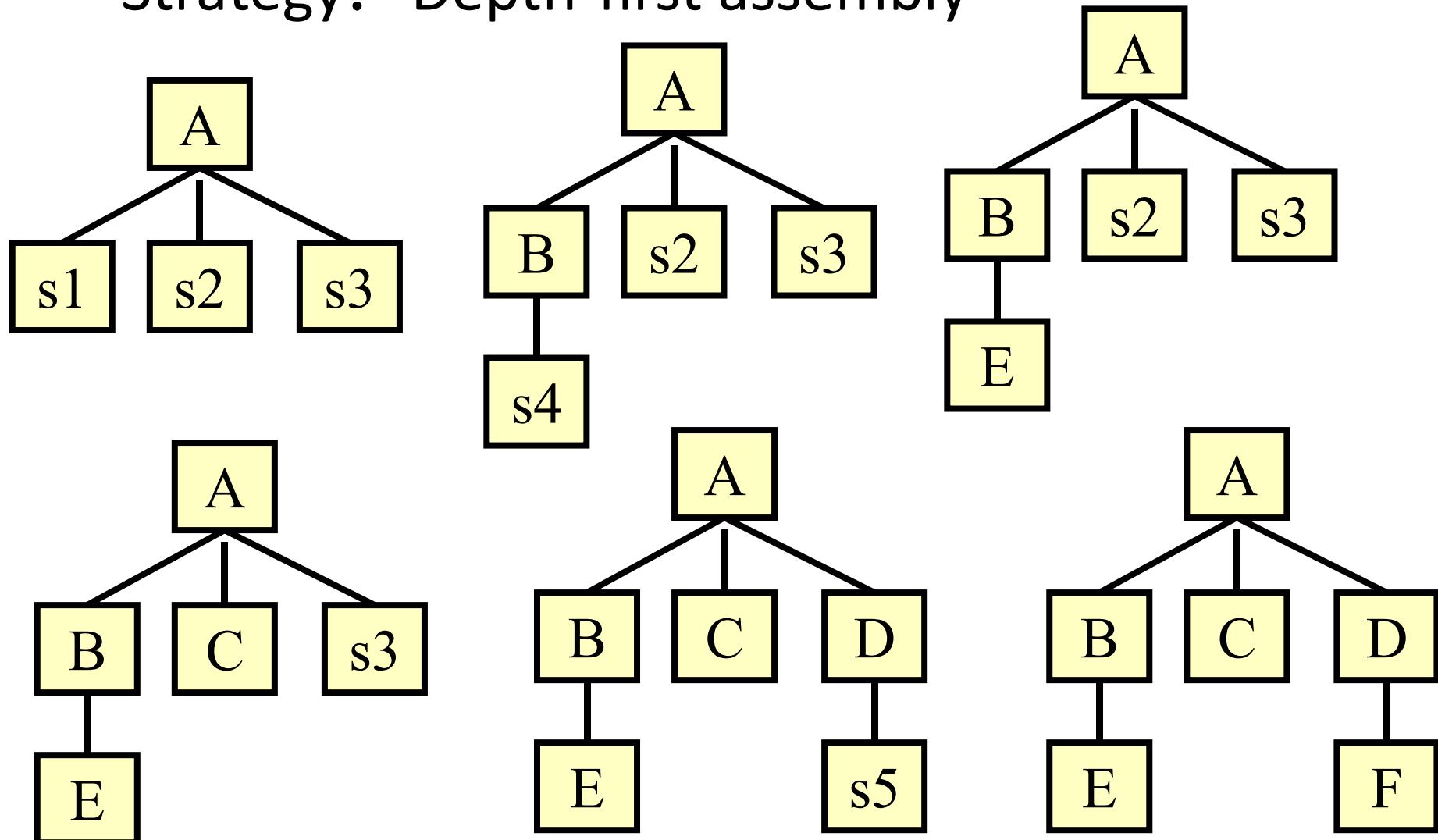
- Scope
 - Existing system with only minor modifications
 - Small systems with adequate unit testing
 - System made from certified high quality reusable components

Integration Testing Strategy

- Top-down integration
 - (1) Focus on the top level components firstly, then gradually test the bottom of components.
 - (2) **Depth-first** and **breadth-first** strategy can be used.
 - (3) Conduct regression testing, exclude possible errors caused by integrated.
 - (4) All modules are integrated into the system then finish testing, otherwise turn to (2) .

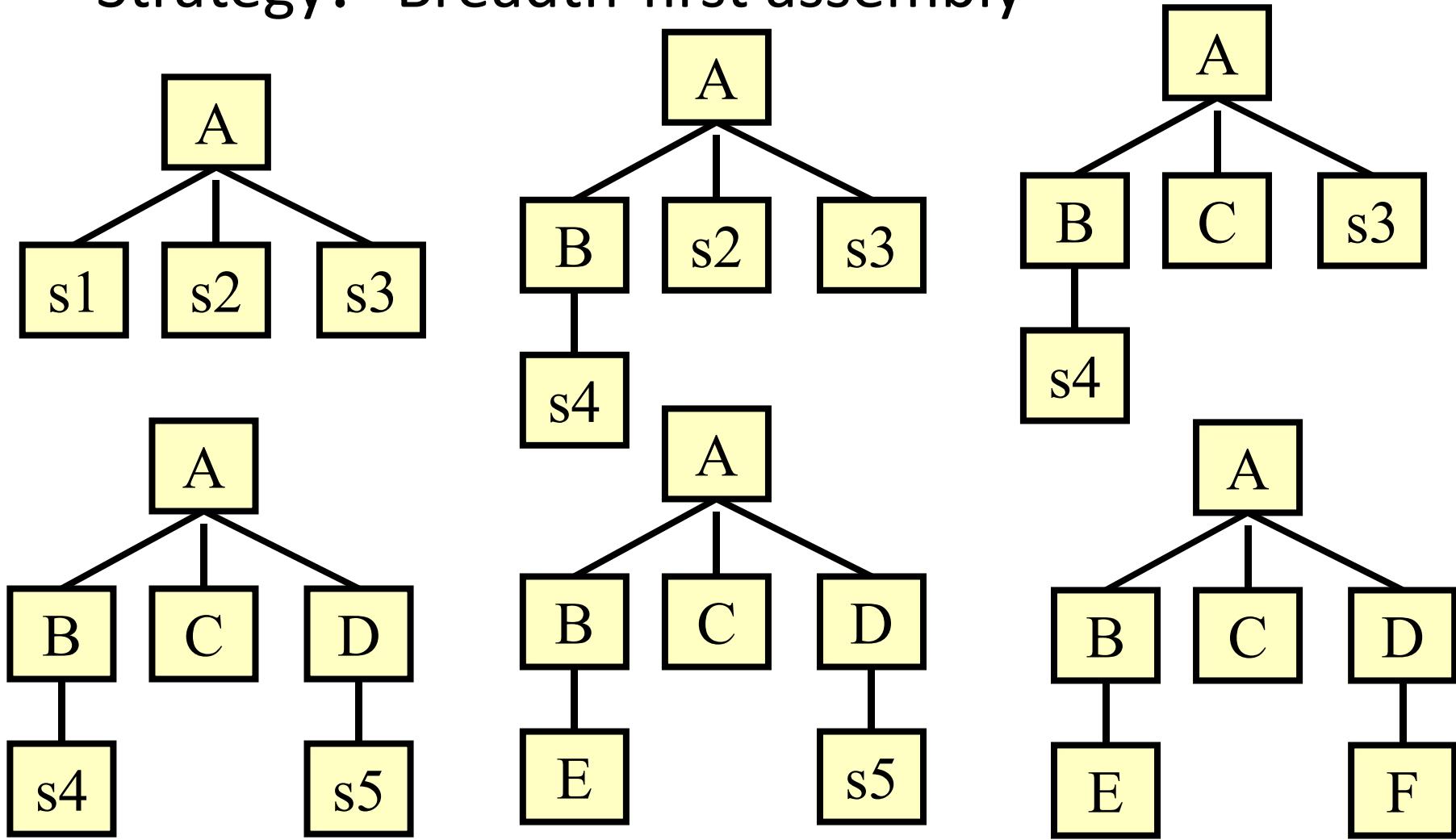
Integration Testing Strategy

- Strategy: Depth-first assembly



Integration Testing Strategy

- Strategy: Breadth-first assembly

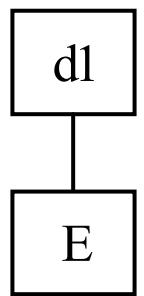


Integration Testing Strategy

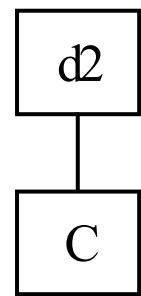
- Bottom-up integration
 - Start from the bottom with minimal dependent components, according to **dependency structure**, layers upward integration to detect the entire system.

Integration Testing Strategy

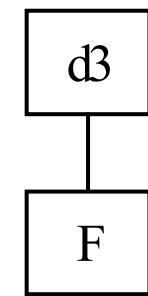
- Bottom-up integration



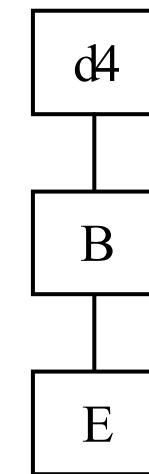
(a)



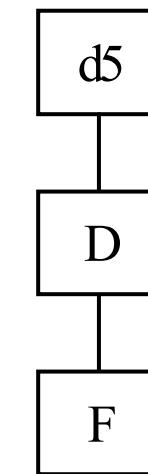
(b)



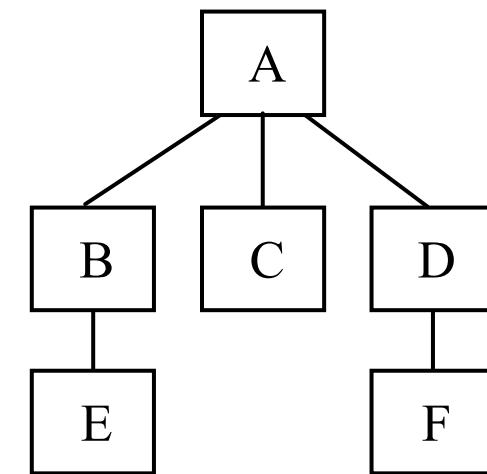
(c)



(d)



(e)



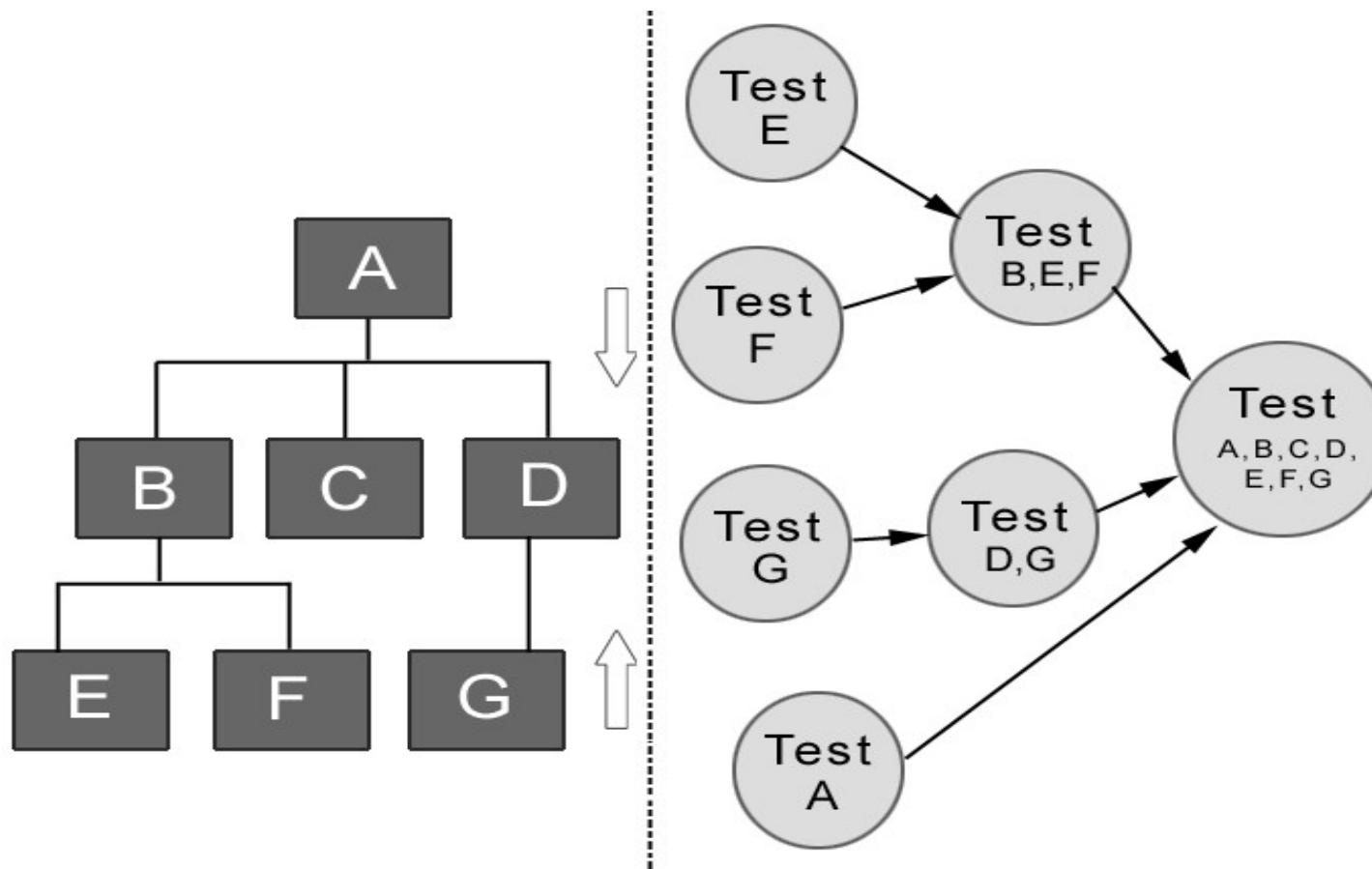
(f)

Integration Testing Strategy

- Sandwich integration
 - Combine the advantage of top-down strategy and bottom-up strategy.

Integration Testing Strategy

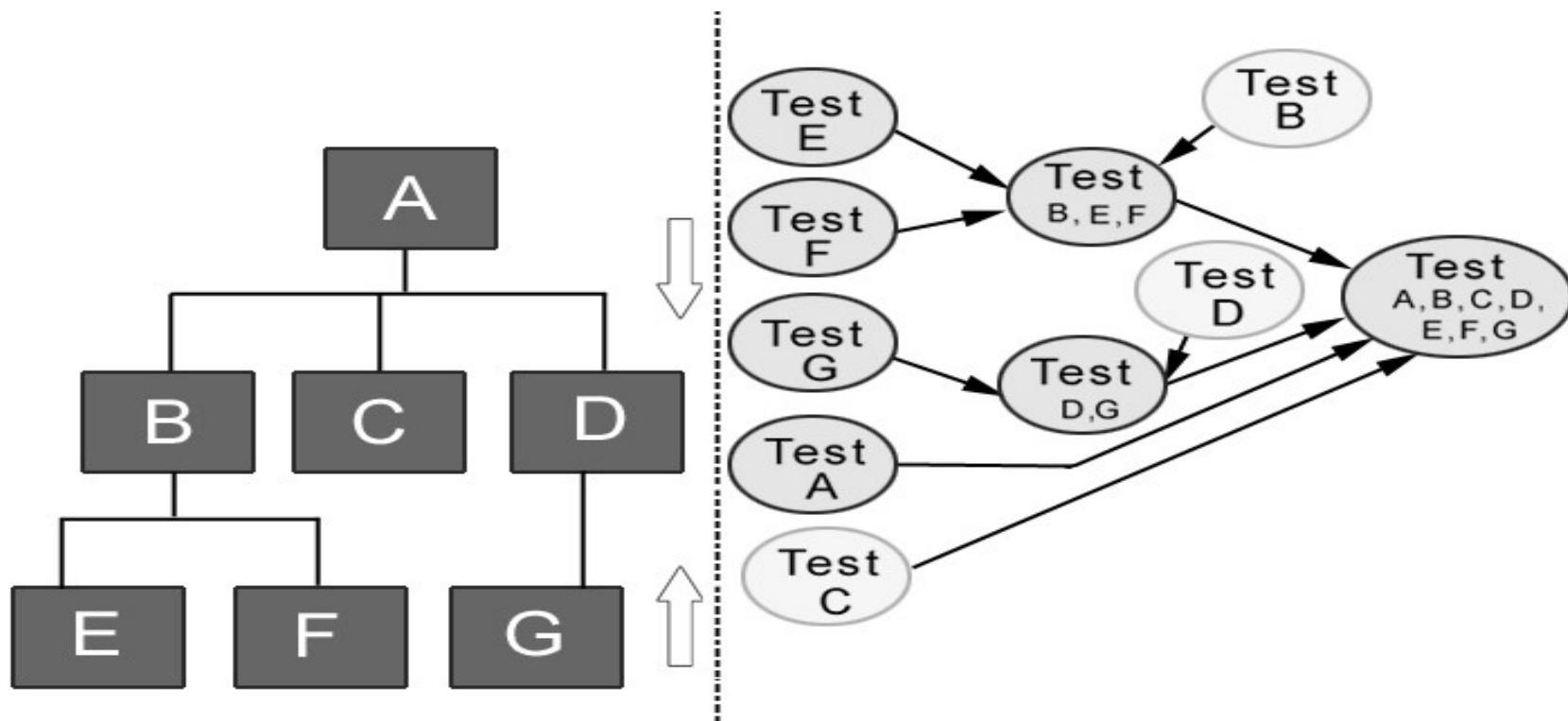
- Sandwich integration



在真正集成之前每一个独立的模块没有完全测试过。

Integration Testing Strategy

- Improved Sandwich integration



保证每个模块得到单独的测试，使测试进行得比较彻底。

Integration Testing Strategy

集成测试策略		
类型	非渐增式集成	渐增式集成
基本方法	先进行单元测试，再将所有模块一起进行集成测试。	把程序划分成小段来构造和测试 自顶向下 自底向上 三明治
特点	需要的用例少，比较简单，效率较高；但不能处理复杂的程序，而且不容易一次成功。	比较容易定位和改正错误，对接口可以进行更彻底测试。

Integration Testing Strategy

渐增式集成			
名称	自顶向下集成	自底向上集成	三明治集成
方法	从主控模块开始，沿着程序控制层次向下移动，逐渐把各模块组合起来。（深度优先或广度优先）	从软件结构最底层的模块开始组装和测试，不需要桩模块。	混合增量式测试策略，综合了自顶向下和自底向上两种集成方法。
优点	可以在早期实现软件的一个完整功能。	可以并行集成，对被测模块可测性要求比自顶向下集成策略低。	桩模块和驱动模块的开发工作都比较小。
缺点	没有底层返回来真实数据流。	驱动模块开发量大，整体设计的错误发现较晚，集成到顶层时将变得越来越复杂。	增加了缺陷的定位难度，目标层在集成前测试不充分。

改进的三明治集成

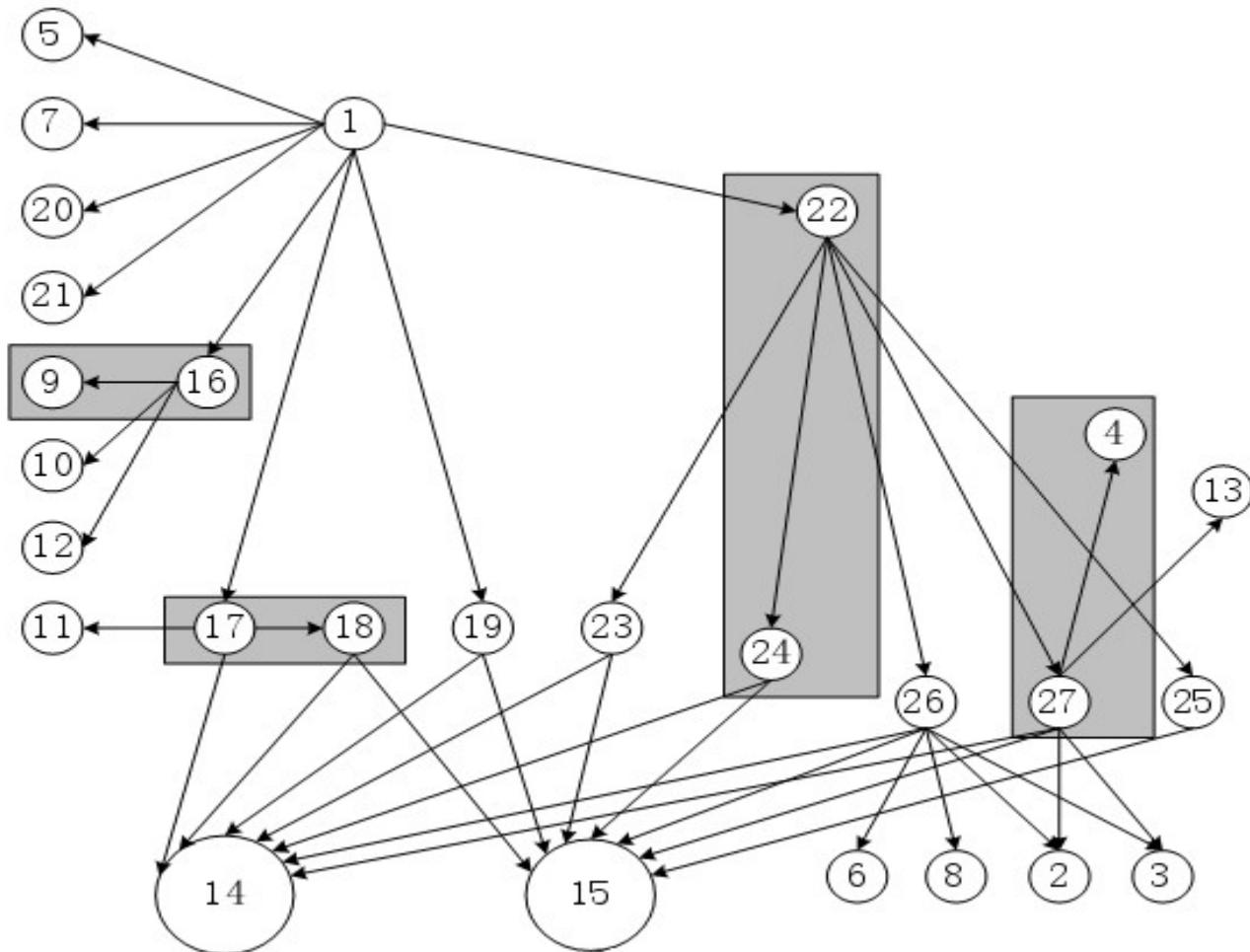
Integration Testing Strategy

- 基于分解的集成—更关注结点
 - Big bang (大爆炸) integration
 - Top-down (自顶向下) integration
 - Bottom-up (自底向上) integration
 - Sandwich (三明治) integration
- 基于调用图的集成—更关注边
 - 成对集成
 - 相邻集成

基于调用图的集成

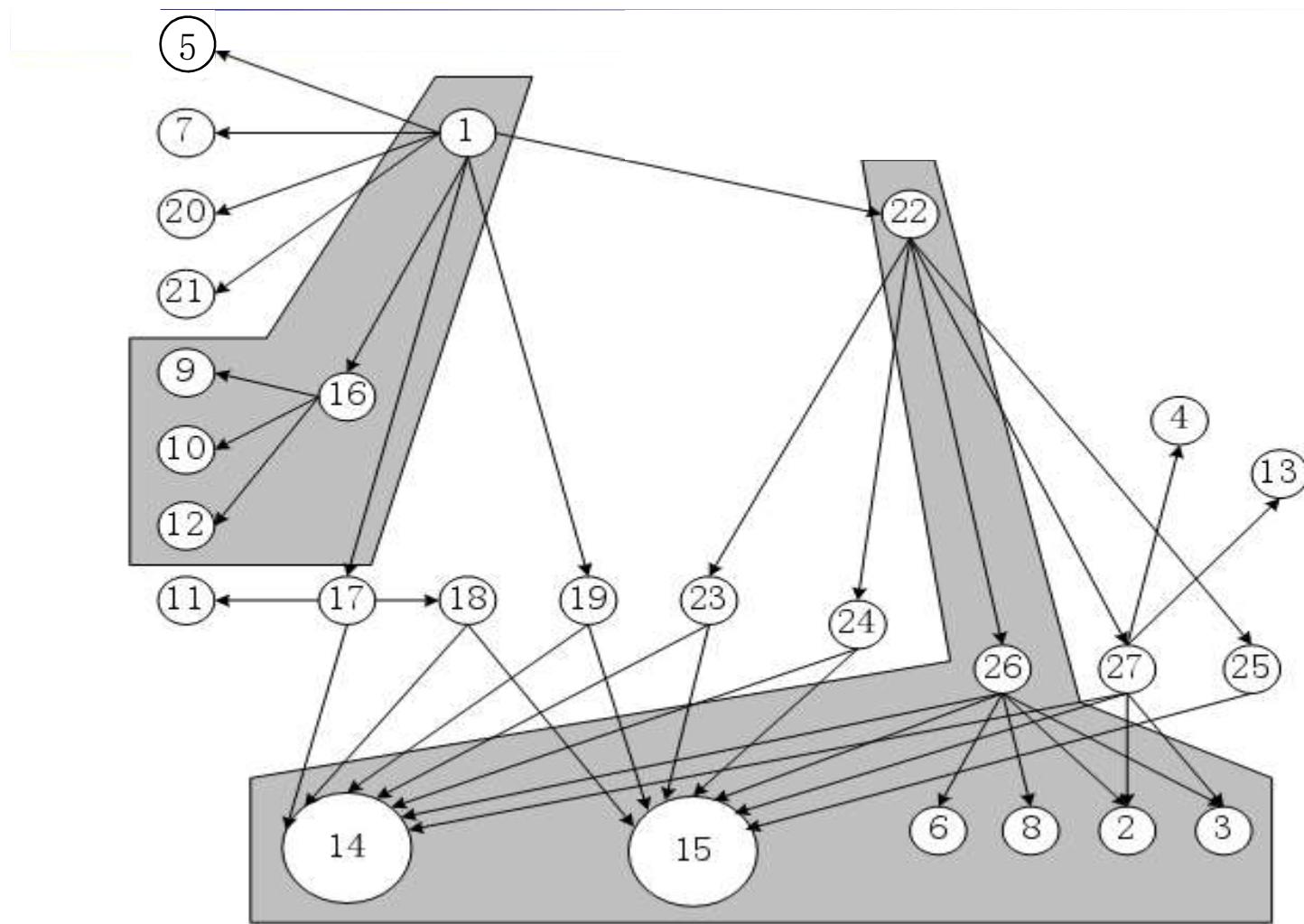
- 成对集成
 - 免除桩/驱动模块的开发工作，使用实际的代码
 - 为避免大爆炸集成，限制在调用图的一对单元上
 - 对调用图中的每条边有一个集成测试过程
- 相邻集成
 - 借用拓扑学中的邻接概念
 - 在有向图中，结点邻居包括所有直接前驱结点和所有直接后继结点
 - 对应结点的桩和驱动模块集合

成对集成示例



40次集成测试会话

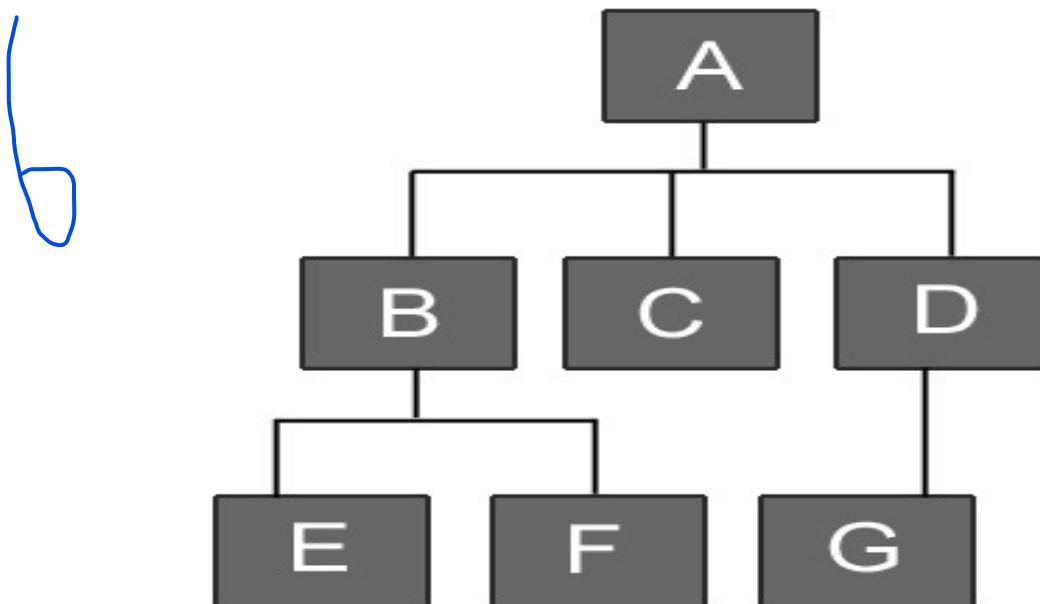
相邻集成示例



节点	前驱	后继
16	1	9, 10, 12
17	1	11, 14, 18
18	17	14, 15
19	1	14, 15
23	22	14, 15
24	22	14, 15
26	22	14, 15, 6, 8, 2, 3
27	22	14, 15, 2, 3, 4, 13
25	22	15
22	1	23, 24, 26, 27, 25
1	-	5, 7, 20, 21, 16, 17, 19, 22

11次集成测试会话

- 成对集成，相邻集成分别需要多少次测试会话？



基于调用图的集成测试的优点

- 免除了驱动器/桩的开发工作
- 接口关系测试充分
- 测试集中于衔接的功能性
- 测试和集成可以并行开始

基于调用图的集成测试的缺点

■ 缺点

- 调用或协作的关系可能是错综复杂的
- 参与者没有被单独测试，要充分测试底层模块较困难
- 特定的调用或协作可能是不完全的
- 缺陷隔离

■ 适用范围

- 尽快论证一个可运行的调用或协作
- 被测系统已清楚定义了模块的调用和协作关系

Integration Testing Strategy

- 基于分解的集成—更关注结点
 - Big bang (*大爆炸*) integration
 - Top-down (*自顶向下*) integration
 - Bottom-up (*自底向上*) integration
 - Sandwich (*三明治*) integration
- 基于调用图的集成—更关注边
 - 成对集成
 - 相邻集成
- 其他集成策略
 - 核心系统先行集成
 - 高频集成

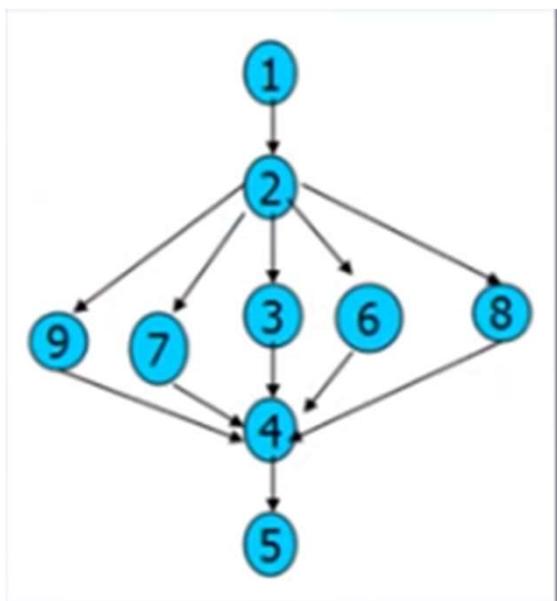
其他集成测试策略

- 基于功能优先级的集成
 - 依据功能的优先级，逐一将某个功能上的各个模块集成起来
 - 核心系统先行集成测试
- 基于进度的集成
 - 将最早获得的模块进行集成，以最大程度保持与开发的并行性
 - 高频集成

核心系统先行集成

- 核心部件先进行集成
- 按各外围软件部件的重要程度逐个集成
- 优点：快速软件开发很有效果，适合较复杂系统的集成测试，能保证一些重要的功能和服务的尽快实现。
- 缺点：应能明确的区分核心软件部件和外围软件部件。

核心系统先行集成



节点12345构成了软件的核心系统，
6789是次要模块。

高频集成

- 高频集成测试是指同步于软件开发过程，每隔一段时间对开发团队的现有代码进行一次集成测试。
- 该集成测试方法频繁地将新代码加入到一个已经稳定的基线中，以免集成故障难以发现，同时控制可能出现的基线偏差。

高频集成

- 使用高频集成测试需要具备一定的条件：
 - 可以持续获得一个稳定的增量，并且该增量内部已被验证没有问题；
 - 大部分有意义的功能增加可以在一个相对稳定的时间间隔（如每个工作日）内获得；
 - 测试包和代码的开发工作必须是并行进行的，并且需要版本控制工具来保证始终维护的是测试脚本和代码的最新版本；
 - 必须借助于使用自动化工具来完成。

高频集成

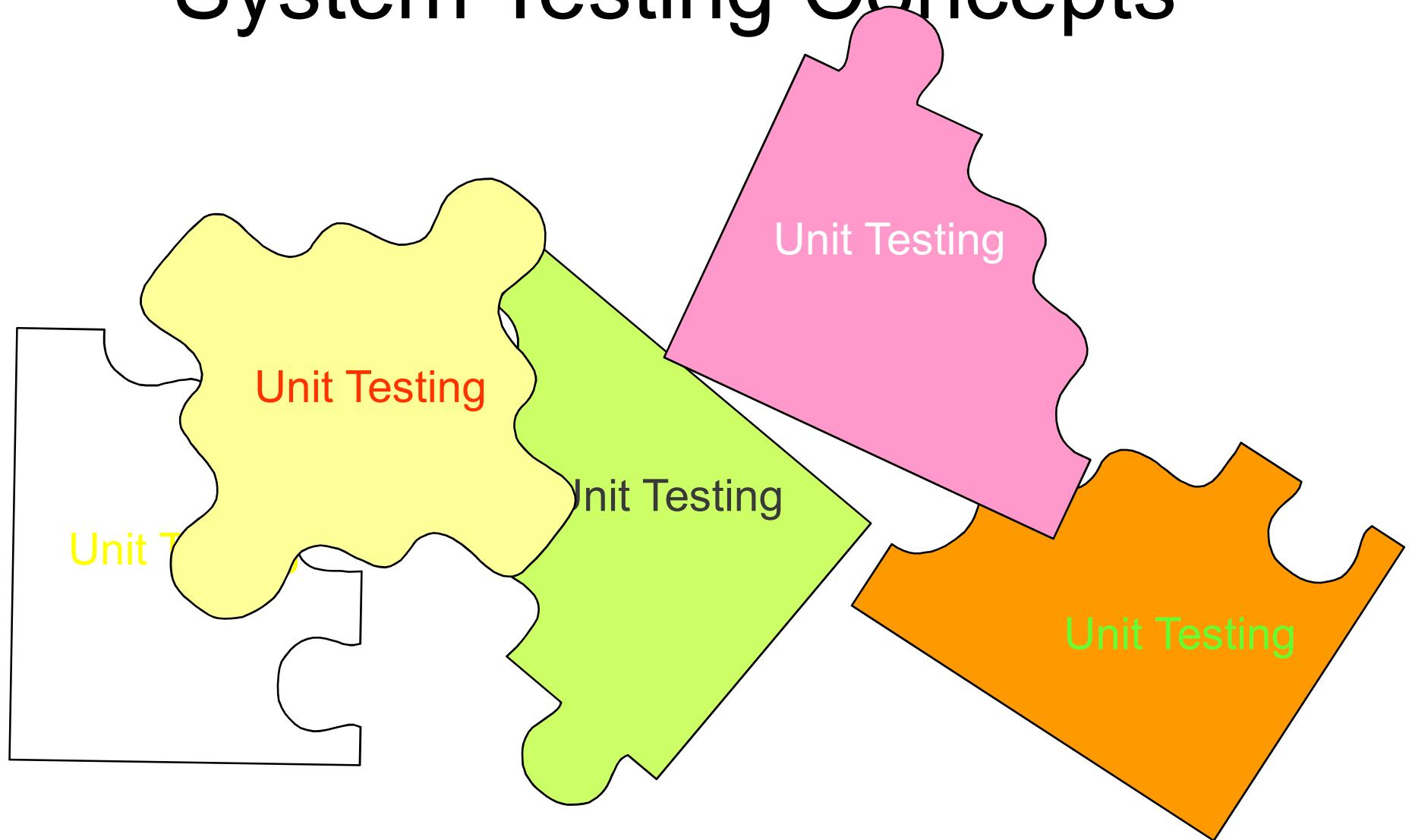
- 优点：能在开发过程中及时发现代码错误，能直观地看到开发团队的有效工程进度。
- 缺点：需要开发和维护源代码和测试包。

Principle of Integration Testing

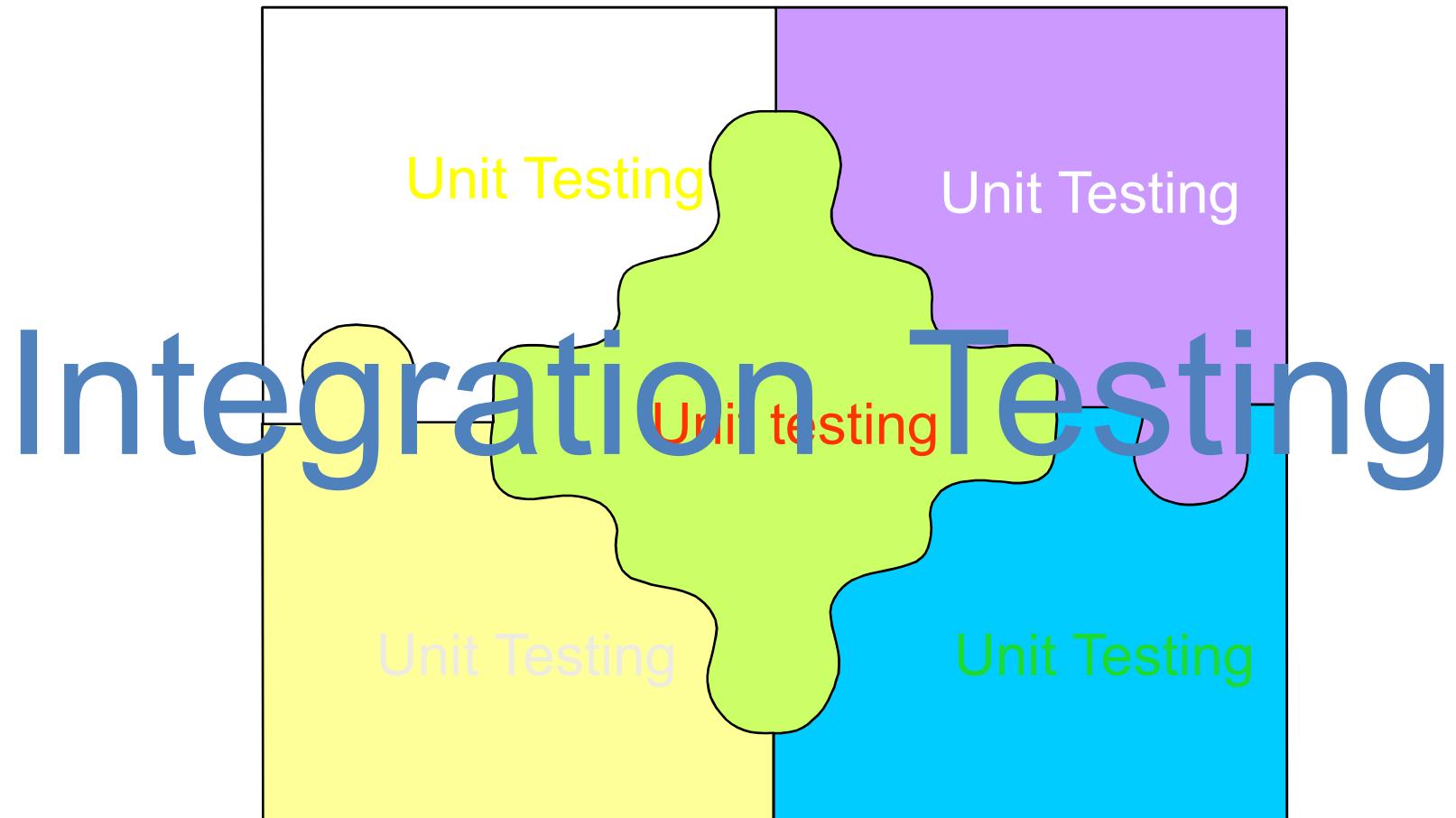
- Key modules must be tested sufficiently. 20-80
- All interfaces must be tested.
- When the interface is changed, all related interfaces should be tested by using regression testing.
- Integration testing is conducted according to plan and prevents random testing.
- Integration testing strategy should integrate the relationship among quality, cost and progress.

- Unit testing
- Integration testing
- **System testing**
- Acceptance testing
- Regression testing

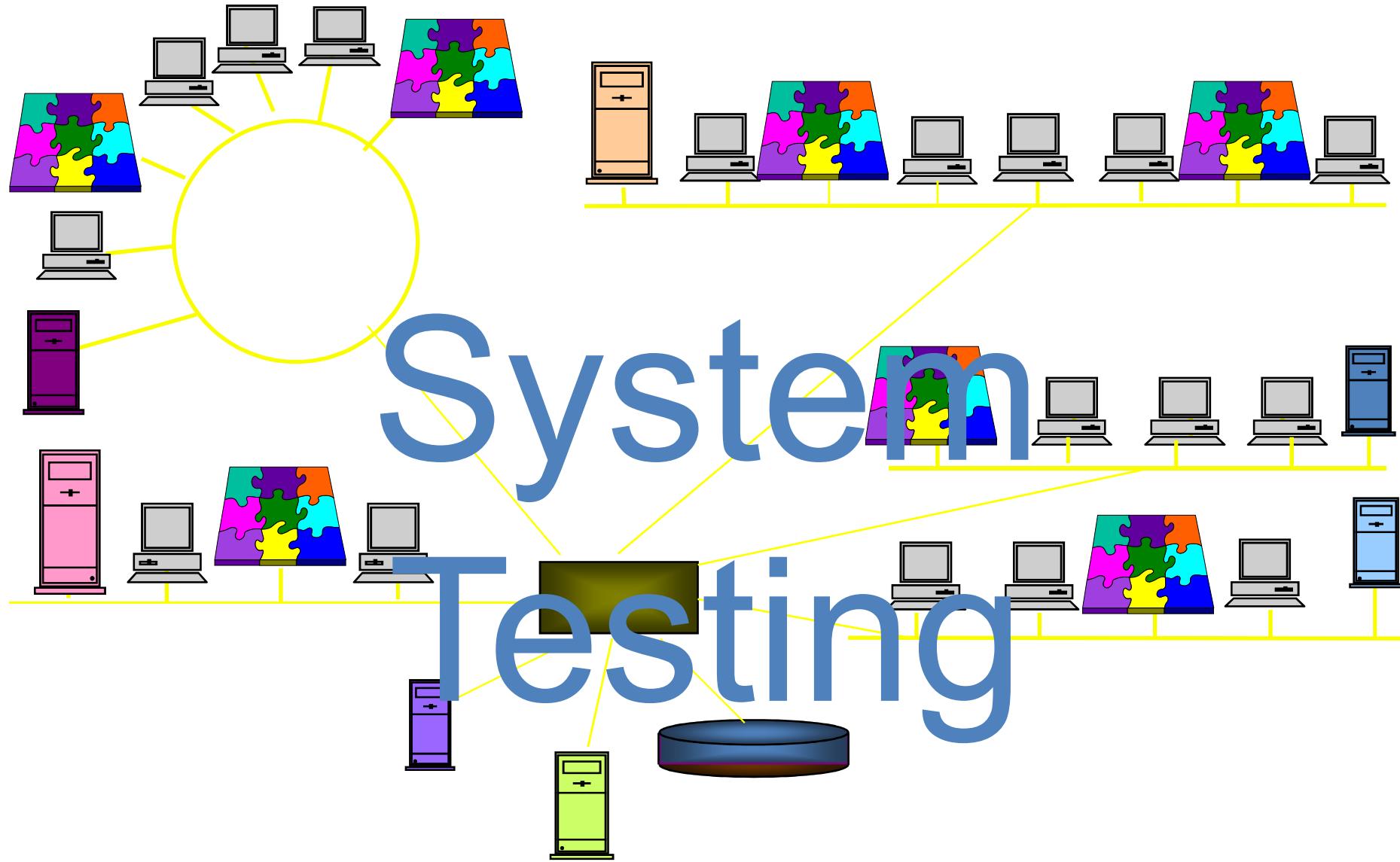
System Testing Concepts



System Testing Concepts



System Testing Concepts



System Testing Concepts

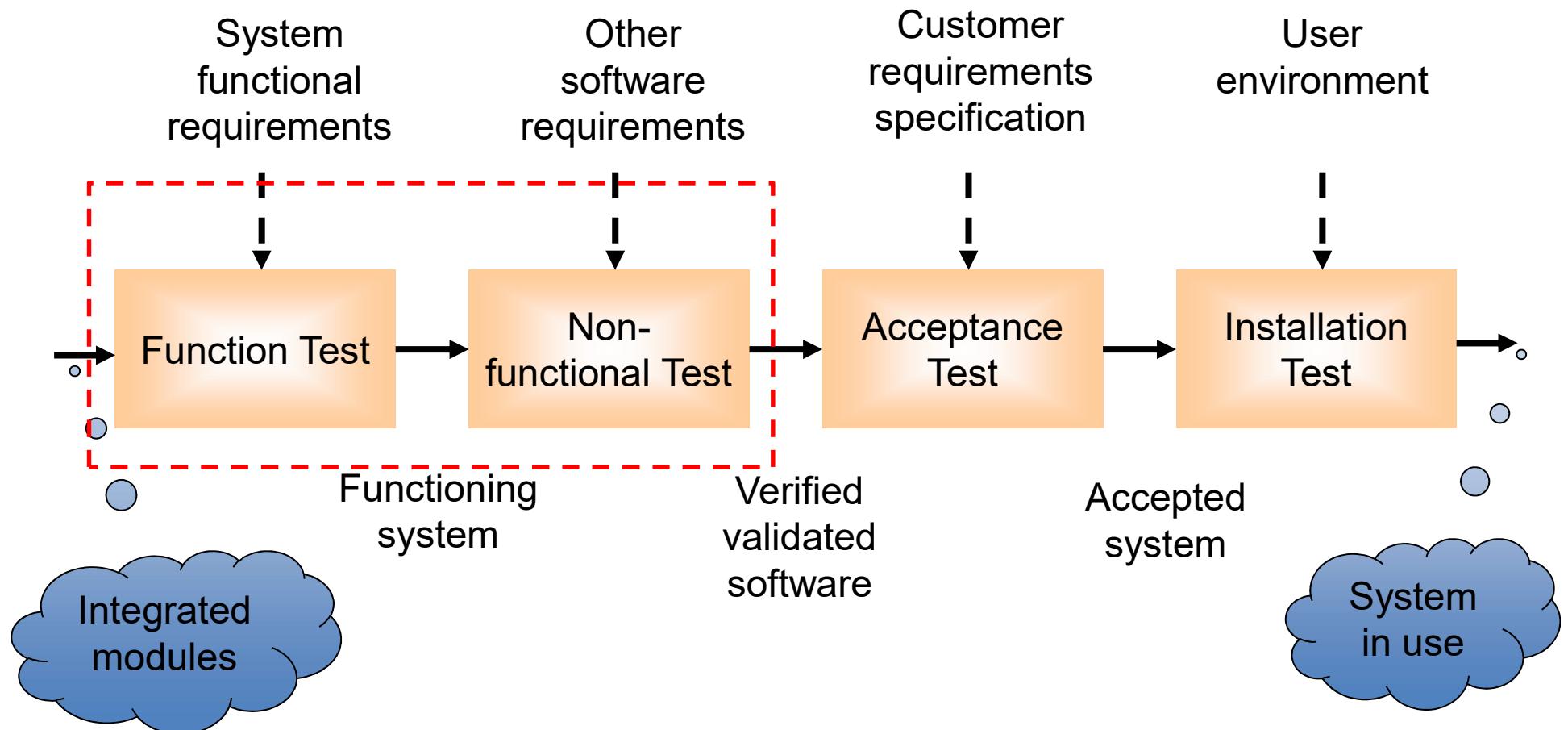
- **Why is system testing necessary?**
 - Some properties only verifiable at system level
 - We may involve users at this level
 - The environment of the system is taken into account

System Testing Concepts

- System testing is the process of testing a fully integrated system to verify if it meets the specified requirements.
- It also determines if the system successfully integrates with the **business procedures** and the **environment**.

System Testing Concepts

- **System testing process**



System Testing Methods

- GUI software testing
- Usability testing
- **Performance testing**
- Compatibility testing
- **Load testing**
- **Volume testing**
- **Stress testing**
- Security testing
- Scalability testing
- **Sanity testing**
- **Smoke testing**
- Exploratory testing
- Ad hoc testing
- **Regression testing**
- Reliability testing
- Recovery testing
- Installation testing
- Maintenance testing
- Accessibility testing

- Unit testing
- Integration testing
- System testing
- Acceptance testing
- Regression testing

Acceptance Testing

- Acceptance testing is a formal testing conducted to determine whether the software **satisfies the acceptance criteria** defined by the customer in the requirements phase of the SDLC.
- **The end user or customer** can conduct acceptance testing to validate whether or not to accept the product.
- 交付测试

Acceptance Testing

- There are two types of acceptance testing:
 - Alpha testing:
 - Alpha testing is a simulated or actual operational testing at an in-house site close to the development team.
 - This type of testing helps evaluate the software to ascertain whether or not it meets all the requirements specified in the requirements analysis phase.
 - Beta testing:
 - Beta testing involves operational testing of the software at a site away from the developers or at the customer site.

Acceptance Testing

- α 测试：早期的、不稳定的软件版本所进行的验收测试，受控的实验室测试
- β 测试：晚期的、更加稳定的软件版本所进行的验收测试，不受控的非实验室测试

Acceptance Testing

- β 测试的局限性：
 - β 测试通常不是专业测试人员，问题往往停留在易用性上；
 - 环境不可控，使用不当引起的问题居多；
 - 为了评价软件或获得软件而参与测试；
 - 反馈信息简单，经常无法重现。
- 众包测试/群体智能的兴起

测试步骤 (Testing life cycle)

- 制定测试计划及验收通过准则，通过客户评审
- 设计测试用例并通过评审
- 准备测试环境与数据，执行测试用例，记录测试结果
- 分析测试结果，根据验收通过准则分析测试结果，作出验收是否通过及测试评价。
 - 测试项目通过；
 - 测试项目没有通过，但存在变通方法，在维护后期或下一个版本改进；
 - 测试项目没有通过，并且不存在变通方法，需要很大的修改；
 - 测试项目无法评估或者无法给出完整的评估。此时须给出原因
- 提交测试报告

- Compare unit testing vs. integration testing vs. system testing vs. acceptance testing:
 - Level
 - Action
 - Actor
 - Method

5W+1H

- Unit testing
- Integration testing
- System testing
- Acceptance testing
- Regression testing